République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université Abderahmane Mira de Béjaïa
Faculté des Sciences Exactes
Département d'Informatique

Support de Cours
Pour la première année LMD en Mathématiques et Informatique (MI).

Module :

Programmation et Structures de Données.

Préparé par

Dr. Samra BOULFEKHAR

Maître de conférences, Catégorie B au Département MI, Faculté des Sciences Exactes, Université Abderahmane Mira de Béjaïa.

Année 2015 – 2016

Avant-propos

Ce polycopié est le support écrit du cours "Programmation et Structures de Données" de la première année Mathématiques et Informatique (MI), Faculté des Sciences Exactes de l'Université A/Mira de Béjaïa, 2014-2015.

Ce cours fait suite au cours "Introduction à l'informatique". Il suppose connu les notions élémentaires comme la programmation itérative, les tableaux, les enregistrements, etc. Il porte sur les notions de bases de la programmation en informatique telles que les algorithmes sur les listes chaînées, les piles, les files, les arbres, la récursivité. Bien sûr, chacune de ces notions ne sera vue que partiellement, mais l'ensemble de ces chapitres se veut représenter un bagage informatique minimal pour tout étudiant MI.

L'objectif du cours est quadruple :

1. Connaitre les différentes Structures de Données (SD) linéaires et arborescentes en informatique,
2. Maîtriser les bases de l'algorithmique sur des structures dynamiques,
3. Voir l'importance de ces SD à travers quelques exemples d'applications,
4. Introduire quelques méthodes, de recherche et de tri, fondamentales en informatique.

Le module "Programmation et Structures de Données" a été intégré dans le nouveau programme lancé dans le cadre du système LMD. La démarche consiste à introduire certaines matières, options, voire spécialités émergeantes. On a conçu le programme détaillé de ce module en s'appuyant sur diverses références : des ouvrages reconnus dans la discipline, mais aussi et surtout des ressources en ligne.

Table des matières

Table des matières i

Introduction générale 2

1 Les Listes 4
- 1.1 Introduction . . . 4
- 1.2 Définition . . . 4
- 1.3 Implémentation de la liste . . . 4
  - 1.3.1 Implémentation contigüe . . . 5
  - 1.3.2 Implémentation chainée . . . 5
- 1.4 Représentation graphique d'une Liste Linéaire Chaînée (LLC) . . . 6
- 1.5 Déclaration d'une LLC . . . 7
- 1.6 Opérations sur les LLC . . . 7
  - 1.6.1 Création d'une liste . . . 8
  - 1.6.2 Parcours d'une liste . . . 9
  - 1.6.3 Insertion d'un élément dans une LLC . . . 10
    - 1.6.3.1 Insertion d'un élément au début d'une LLC . . . 10
    - 1.6.3.2 Insertion d'un élément au milieu d'une LLC . . . 11
    - 1.6.3.3 Insertion d'un élément à la fin d'une LLC . . . 12
  - 1.6.4 Suppression d'un élément d'une LLC . . . 13
    - 1.6.4.1 Suppression du premier élément d'une LLC . . . 14
    - 1.6.4.2 Suppression d'un élément se trouvant au milieu d'une LLC . . . 15
    - 1.6.4.3 Suppression d'un élément se trouvant à la fin d'une LLC . . . 16
- 1.7 Exercice . . . 17

2 Les piles 18
- 2.1 Introduction . . . 18
- 2.2 Définition . . . 18
- 2.3 Implémentation des piles . . . 19
  - 2.3.1 Implémentation par des tableaux . . . 19
  - 2.3.2 Implémentation par des LLCs . . . 19
- 2.4 Représentation graphique d'une pile . . . 19
- 2.5 Opérations sur les piles . . . 20
  - 2.5.1 Pile vide ? . . . 20
  - 2.5.2 Empiler un élément sur une pile . . . 21
  - 2.5.3 Sommet de la pile . . . 21
  - 2.5.4 Dépiler un élément d'une pile . . . 22

2.5.5 Vider une pile . . . 22
2.6 Applications importantes des piles . . . 22
2.6.1 Les appels récursifs . . . 23
2.6.2 L'évaluation d'une expression arithmétique . . . 24
2.7 Exercice . . . 25

3 Les files **26**
3.1 Introduction . . . 26
3.2 Définition . . . 26
3.3 Implémentation d'une file . . . 26
3.4 Représentation graphique d'une file . . . 27
3.5 Opérations sur les files . . . 27
3.5.1 File vide ? . . . 28
3.5.2 Le premier et le dernier élément de la file . . . 28
3.5.3 Enfiler un élément dans une file . . . 29
3.5.4 Défiler un élément d'une file . . . 29
3.5.5 Vider une file . . . 30
3.6 Applications importantes des files . . . 31
3.7 Exercice . . . 31

4 Les arbres **32**
4.1 Introduction . . . 32
4.2 Définition . . . 32
4.3 Représentation graphique d'un arbre . . . 33
4.4 Arbre binaires . . . 35
4.4.1 Notion d'un arbre n-aire . . . 35
4.4.2 Notion d'un arbre binaire . . . 35
4.4.3 Arbres binaires particuliers . . . 36
4.4.3.1 Arbre binaire complet . . . 36
4.4.3.2 Arbre binaire localement complet . . . 36
4.4.3.3 Arbre binaire quasi-complet ou parfait . . . 37
4.4.3.4 Arbre binaire dégénéré ou filiforme . . . 37
4.4.3.5 Arbre binaire peigne . . . 37
4.4.4 Transformation d'un arbre n-aire en un arbre binaire . . . 38
4.4.5 Occurrence et numérotation hiérarchique . . . 38
4.4.6 Implémentation d'un arbre binaire . . . 40
4.4.7 Opérations de base sur les arbres binaires . . . 41
4.4.8 Parcourir un arbre . . . 41
4.4.8.1 Calculer le degré d'un arbre . . . 43
4.4.8.2 Savoir si un nœud est une feuille . . . 44
4.5 Applications importantes des arbres . . . 44
4.5.1 Le calcul arithmétique . . . 44
4.5.2 Le codage de Huffman . . . 44
4.6 Exercice . . . 45

5 Etude de quelques méthodes de tri et de recherche 46
5.1 Introduction 46
5.2 Méthodes de tri 46
5.2.1 Tri par sélection 47
5.2.2 Tri par transposition "bulle" 48
5.2.3 Tri par insertion 49
5.3 Méthodes de recherche 50
5.3.1 Recherche séquentielle 51
5.3.2 Recherche dichotomique 52
5.4 Exercice 53

Conclusion générale 54

Avant-propos

Ce polycopié est le support écrit du cours "Programmation et Structures de Données" de la première année Mathématiques et Informatique (MI), Faculté des Sciences Exactes de l'Université A/Mira de Béjaïa, 2014-2015.

Ce cours fait suite au cours "Introduction à l'informatique". Il suppose connu les notions élémentaires comme la programmation itérative, les tableaux, les enregistrements, etc. Il porte sur les notions de bases de la programmation en informatique telles que les algorithmes sur les listes chaînées, les piles, les files, les arbres, la récursivité. Bien sûr, chacune de ces notions ne sera vue que partiellement, mais l'ensemble de ces chapitres se veut représenter un bagage informatique minimal pour tout étudiant MI.

L'objectif du cours est quadruple :

1. Connaitre les différentes Structures de Données (SD) linéaires et arborescentes en informatique,
2. Maîtriser les bases de l'algorithmique sur des structures dynamiques,
3. Voir l'importance de ces SD à travers quelques exemples d'applications,
4. Introduire quelques méthodes, de recherche et de tri, fondamentales en informatique.

Le module "Programmation et Structures de Données" a été intégré dans le nouveau programme lancé dans le cadre du système LMD. La démarche consiste à introduire certaines matières, options, voire spécialités émergeantes. On a conçu le programme détaillé de ce module en s'appuyant sur diverses références : des ouvrages reconnus dans la discipline, mais aussi et surtout des ressources en ligne.

Introduction générale

UNE structure de donnée (SD) est une famille d'informations qu'un programme peut décrire, créer et manipuler. Définir une SD, pour un ensemble des données, consiste à :

1. Définir une certaine organisation de ces données,
2. Définir les primitives ou les fonctions d'accès et de manipulation de ces données,
3. Définir une représentation interne de ces données.

Une SD peut être caractérisée par les opérations fondamentales que l'ont peut faire dessus. La connaissance des propriétés des structures de données permet de guider le choix des représentations, notamment dans le souci d'optimisation des performances des solutions obtenues. Il existe deux façons de représenter les structures de données : par tableaux et par chaînage dynamique. L'emploi de tableaux ne pose pas de difficulté particulière et présente l'avantage de mettre en évidence la façon dont la mémoire est gérée de façon interne. Nous ne la traitons pas dans ce document et laissons ce soin à l'étudiant.

Dans les exemples traités dans ce cours, nous utilisons le chaînage dynamique pour la mise en œuvre de structures linéaires (listes, piles et files) et des structures hiérarchiques (arbres et plus particulièrement arbres binaires).
Pour chacune de ces structures de données, nous commencerons par présenter les différentes manières de leurs modélisations. Ensuite, nous détaillerons en langage algorithmique les principales opérations qui peuvent être appliquées sur ces structures. Enfin, pour certaines d'entre elles, nous développons quelques exemples d'utilisation.

Ce support de cours "Programmation et Structures de Données" s'intéresse, comme on a dit auparavant, à la présentation des différentes structures de données ainsi que quelques algorithmes de tri et de recherche.

Ce support de cours, s'articulera autour de cinq chapitres.

Le chapitre I discute les concepts des listes chaînées définies comme collections linéaires de données appelés cellules qui se pointent les unes aux autres par des pointeurs. En plus, il explique les différentes opérations relatives.

Dans le chapitre II, la structure de donnée Pile sera présentée. Cette structure est une liste particulière dans laquelle les opérations d'insertion et de suppression s'effectuent à une extrémité de la liste.

Le chapitre III introduit une autre structure de données linéaire appelée File. Cette structure de données aussi est inspirée de la structure listes. Dans les files, les opérations d'insertion s'effectuent à une extrémité et les opérations de suppression s'effectuent à l'autre extrémité.

Le chapitre IV présente l'une des plus importantes structures de donnée non linéaires arbre. Dans ce chapitre, les divers termes liés à cette structure vont être définis, les deux types d'arbres (n-aire et binaire) vont être éplorés et les différentes opérations applicables sur un arbre binaire vont être détaillées.

Le chapitre V est dédié aux principales méthodes de tri et de recherche. Pour chaque méthode à présenter, nous donnerons son principe ainsi que l'algorithme correspondant.

Enfin, ce support de cour se clôturera par une conclusion générale et une liste de références.

Chapitre 1

Les Listes

1.1 Introduction

Une structure de données séquentielle est un ensemble fini de données organisées de telle sorte que leurs traitements se font d'une façon séquentielle. La structure la plus répondue et la plus simple est la liste souvent dite liste linéaire vu que ses éléments sont traités linéairement. En effet, les listes linéaires sont vues comme une généralisation des structures étudiées précédemment : tableaux (voir chapitre 5, module Initiation à l'informatique). Elles permettent de représenter un ensemble de données, d'insérer ou de supprimer des éléments sans qu'il soit nécessaire d'en déplacer d'autres comme les tableaux.

1.2 Définition

Une Liste linéaire **L** est une suite finie d'éléments repérés par leur rang dans la liste : $L =< e_1, e_2, e_3, ...e_n >$. L'ordre des éléments dans une liste est fondamental. Cet ordre n'est pas sur les éléments mais sur les places des éléments. L'ordre sur les places indique qu'il y a une place qui est la 1ere de toutes notée **tête(L)**. Le parcourt d'une liste nécessite la connaissance de tête(L). Ce parcourt devra s'arrêter à une position appelée fin de liste notée par le mot **Nil**.

1.3 Implémentation de la liste

Il y a deux modes de représentation d'une liste :

1.3.1 Implémentation contigüe

Une liste est représentée, dans ce cas, par un tableau à une seule dimension et une variable contenant la longueur de la liste, donc un couple <**tableau, entier**>. Le tableau en tant que tel n'est jamais parcouru comme un tableau normal dans un sens ou dans l'autre. C'est toujours la liste qui est lue et utilisée dans l'ordre où sont rangés les éléments qui la composent. Cette représentation permet une **implémentation simple** de la liste. Cependant, les **opérations d'insertion et de suppression posent le problème** suivant :
L'insertion d'un élément à la k^{ieme} position nécessite un déplacement de tous les éléments successeurs(la même chose pour la suppression). En plus, il faudrait connaitre au préalable la taille maximale de la liste car l'utilisation des tableaux implique que l'allocation de l'espace se fait tout au début d'un traitement, c'est à dire que l'espace est connu à la compilation (il faut **prévoir une taille maximale dès le départ**). On est donc contraint à utiliser une autre forme d'allocation dont on alloue l'espace au fur et à mesure de l'exécution du programme.

1.3.2 Implémentation chainée

C'est l'implémentation la plus naturelle et la plus utilisée. Dans cette représentation, chaque élément de la liste est stocké dans une cellule et les cellules sont reliées par des pointeurs. Chaque cellule contient, en plus de la donnée à traiter, l'adresse de la cellule suivante. A chaque cellule est associée une adresse mémoire. Les Listes Linéaires Chaînées (**LLC**) font appel à la notion de variable dynamique. Cette dernière peut y être créée ou détruite (c'est-à-dire, on lui alloue un espace mémoire à occuper et qu'il sera libéré après son utilisation). L'accès à sa valeur se fait à l'aide d'un pointeur.

- Un pointeur est une variable dont la valeur est une adresse mémoire (voir chapitre 2, module Initiation à l'informatique). Un pointeur, noté **P**, pointe sur une variable dynamique notée $P^\wedge$.
- Le type de base est le type de la variable pointée.
- Le type du pointeur est l'ensemble des adresses des variables pointées du type de base. Il est représenté par le symbole ^ suivi de l'identificateur du type de base.

Exemple :

La variable pointeur **P** pointe sur l'espace mémoire $P^\wedge$ d'adresse **3**. Cette cellule mémoire contient la valeur "Exemple" dans le premier champ et la valeur spéciale **Nil** dans le deuxième champ. Ce champ servira à indiquer quel est l'élément suivant lorsque la cellule fera partie d'une LLC. La valeur Nil indique qu'il n'y a pas d'élément suivant. Les LLC entraînent l'utilisation

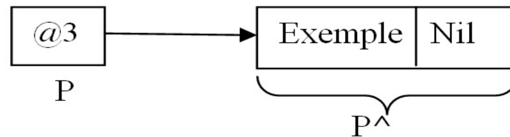


FIGURE 1.1 – Exemple sur un pointeur et une variable.

de procédures d'allocation et de libération dynamiques de la mémoire. Ces procédures sont les suivantes :

- **Allouer(P)** : réserve un espace mémoire $P^\wedge$ et affecte à P l'adresse de cet espace mémoire. On alloue un espace mémoire pour un élément sur lequel pointe P.
- **libérer(P)** : libère l'espace mémoire qui était occupé par l'élément à supprimer $P^\wedge$ sur lequel pointe P.

Remarque importante :

Dans le reste de ce cours, on considérera seulement la représentation chainée pour modéliser les différentes SD à étudier.

1.4 Représentation graphique d'une Liste Linéaire Chaînée (LLC)

Avant de présenter les différents algorithmes manipulant une LLC, il est utile de montrer un schéma représentant graphiquement l'organisation des éléments de la LLC. Dans l'exemple (la figure 1.2) L est une LLC de quatre cellules. Chaque cellule comporte deux parties : la première partie représente la donnée à traiter, notée **info**, et la deuxième partie indique l'adresse de la cellule suivante, notée **suivant**. Ce schéma met en évidence un certain nombre de points :

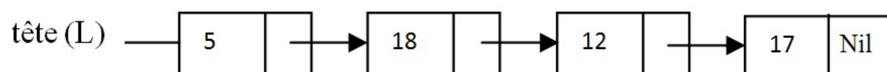


FIGURE 1.2 – Représentation graphique d'une LLC.

– Une LLC est caractérisée par l'adresse de son premier élément tête(L),

- Nil constitue l'adresse qui ne pointe sur aucune cellule. Elle est utilisée pour marquer la fin de la liste,
- Une LLC vide est une liste qui ne contient aucun élément, elle est représentée par Nil,
- Si la place P est référencée par le pointeur P dans L, le suivant de P sera référencé par P^.suivant. Dans l'exemple, si P occupe la position 2 (la cellule qui contient la valeur 18), P^.suivant pointe sur la cellule 3 (la cellule qui contient la valeur 12),
- Pour acceder à la valeur stockée dans la cellule **C** de la liste L, il suffit d'écrire C^.info. Dans l'exemple, si **C** représente la cellule 2, alors C^.info donne la valeur 18.

1.5 Déclaration d'une LLC

Il faut tout d'abord commencer par définir le type de variable pour chaque élément de la LLC. En langage algorithmique, ceci se fait comme suit :

```
type liste =^cellule
    cellule = enregistement
        info : type_info; /*n'importe quel type*/
        suivant : liste;
    fin;
```

Une fois le type *liste* est défini, on peut déclarer une variable de type pointeur sur *liste* de la manière suivante :

```
var P, tete : liste;
```

1.6 Opérations sur les LLC

Dans cette section, on présentera quelques opérations de base sur les LLC. Parmi ces opérations, nous citons :

- La création d'une LLC,
- Le parcours d'une LLC,
- L'insertion d'un élément dans une LLC,
- La suppression d'un élément d'une LLC.

Remarque

Lors la création, attention à bien **initialiser la tête de la liste à Nil**, sinon la chaine n'aura pas de buttoir finale et il ne sera pas possible de repérer sa fin. Une LLC est connue par l'adresse de son premier élément. Si cette adresse n'est pas mémorisée, la liste disparait. Donc, il faut toujours avoir **l'adresse de la tête mémorisée.**

1.6.1 Création d'une liste

La création d'une LLC consiste à créer et enchaîner, au fur et à mesure, les éléments constituant la LLC de telle sorte que le pointeur tête pointe sur le premier élément. Le premier élément pointe sur le second et ainsi de suite. Le dernier élément ne pointe sur aucun autre élément, donc, sa partie suivant pointe sur Nil.

Exemple 1 :

Création d'une LLC composée de 2 éléments entiers.

```
algorithme creation2;
type liste =^cellule
        cellule = enregistement
            info : entier;
            suivant : liste;
        fin;
var L, P, Q : liste;
debut
|    L ← Nil;                /* Pour l'instant la liste est vide*/
|    Allouer(P);             /* Réserver un espace mémoire pour le premier élément */
|    Lire (P^.info);         /* Stocker dans l'Info de l'élément pointé par P la valeur saisie */
|    P^.suivant ← Nil;       /* Il n'y a pas d'élément suivant */
|    L ← P;                  /* Le pointeur Tete pointe sur le premier élément P */
|                  /* Il faut ajouter le 2ieme élément */
|    Allouer(Q);             /* Réserver un espace mémoire pour le second élément */
|    Lire (Q^.info);         /* Stocker dans l'Info de l'élément pointé par P la valeur saisie */
|    P^.Suivant ← Q;         /* Relier le premier élément avec le deuxième élément */
|    Q^.Suivant ← Nil;       /* Mettre Nil dans la partie adresse du dernier élément*/
fin.
```

Algorithme I.1. Création d'une LLC avec 2 cellules.

Exemple 2 :

Création d'une LLC composée de plusieurs éléments entiers.

```
algorithme creation_plusieurs_elets;
type liste =^cellule;
        cellule = enregistement
            info : entier;
            suivant : liste;
        fin;
var L, P, Q : liste; reponse : chaine[3];
```

```
debut
|     L ← Nil ;                  /* Pour l'instant la liste est vide*/
|     Allouer(P) ;                 /* Réserver un espace mémoire pour le premier élément */
|     Lire (P^.info) ;           /* Stocker dans l'Info de l'élément pointé par P la valeur saisie */
|     P^.suivant ← Nil ;      /* Il n'y a pas d'élément suivant */
|     L ← P ;                    /* Le pointeur Tete pointe maintenant sur P */
|     Repeter
|        | Allouer(Q) ;  /* Réserver un espace mémoire pour l'élément suivant */
|        | Lire (Q^.info) ;  /* Stocker dans l'Info de l'élément pointé par Q la valeur saisie */
|        | Q^.suivant ← Nil ;  /* Mettre Nil dans la partie adresse du dernier élément*/
|        | P^.suivant ← Q ;  /* Relier l'élément précédant avec l'élément suivant */
|        | P ← Q ;  /* Le pointeur P pointe sur Q*/
|        | Ecrire ("Voulez vous créer un autre élément") ;
|        | Lire (reponse) ;
|     Jusqu'à (réponse ='NON') ;
fin.
```

Algorithme I.2. Création d'une LLC avec plusieurs cellules.

1.6.2 Parcours d'une liste

Pour parcourir une LLC, il suffit avec un pointeur de prendre successivement l'adresse de chaque cellule. Au départ, le pointeur prend l'adresse du premier élément qui est l'adresse de la tête de liste, ensuite, il se déplace vers l'adresse du suivant et ainsi de suite.

Exemple 1 :

Parcourir la liste L de §I.4, en affichant le contenu du champ info.

```
algorithme parcours ;
type liste =^cellule
       cellule = enregistement
          info : entier ;
          suivant : liste ;
       fin ;
var L, P : liste ;
debut
|    si L = Nil alors
|     | ecrire ("L est vide") ;
|     sinon
|     | P ← L ; /*Mémoriser l'adresse de la tête*/
```

```
|       | TQ( P< >Nil) faire /* Parcourir L jusqu'au dernier élément*/
|       |       | Ecrire (P^.info) ; /*Afficher le contenu du champ Info*/
|       |       | P← P^.suivant ; /* Passer d'un élément à l'autre*/
|       | finTQ ;
|       | finsi ;
fin.
```

Algorithme I.3. Parcours d'une LLC.

1.6.3 Insertion d'un élément dans une LLC

Le fait d'insérer plutôt qu'ajouter un élément dans la liste suppose un classement. Il faut un critère d'insertion, pourquoi ici et pas là ? Par exemple, nous allons insérer nos éléments de façon à ce que les valeurs entières soient classées en ordre croissant. Trois cas sont possibles :

- L'insertion au début de la LLC,
- L'insertion au milieu de la LLC,
- L'insertion à la fin de la LLC.

1.6.3.1 Insertion d'un élément au début d'une LLC

L'insertion d'un élément en tête de la liste est l'opération essentielle sur les LLC, est appelée aussi constructeur. Pour réaliser cette opération, il suffit de faire pointer le champ suivant du nouvel élément de tête vers l'ancienne LLC, qui devient ainsi la tête de la nouvelle (voir la figure I.3). Il y a deux actions, dans cet ordre, à réaliser (voir l'algorithme I.4) :

- Créer le nouvel élément.
- Créer un lien entre le nouvel élément et la tête de la LLC de telle sorte que la tête soit le suivant du nouvel élément et le nouvel élément devient le premier élément de la LLC.

Exemple :

```
algorithme Ajout_debut ;
type liste =^cellule ;
        cellule = enregistement
            info : entier ;
            suivant : liste ;
        fin ;
var L, nouveau : liste ;
```

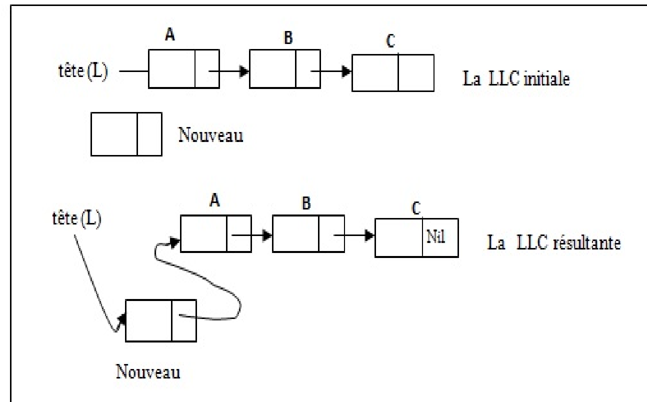


FIGURE 1.3 – L'insertion d'un nouvel élément au début d'une LLC.

debut
| Allouer(nouveau) ; /* Réserver un espace mémoire pour le premier élément*/
| Lire (*nouveau*^.info) ; /* Stocker dans l'Info de l'élément pointé par nouveau la valeur saisie*/
| *nouveau*^.suivant ← L ; /* Créer un lien entre le nouvel élément et la tête de la LLC */
| L ← nouveau ; /* Le pointeur L pointe sur nouveau, et nouveau devient le premier élément de L */
fin.

Algorithme I.4. Insérer une nouvelle cellule au début d'une LLC.

1.6.3.2 Insertion d'un élément au milieu d'une LLC

Pour ce faire, il faut chercher la bonne place et conserver l'adresse de l'élément qui précède. Pour repérer l'élément précédent, la liste est parcourue tant que l'élément courant n'est pas nulle et que le bon emplacement n'est pas encore trouvé. Nous avons besoin de deux pointeurs : un pour l'élément courant et un autre pour l'élément précédent. A chaque itération, avant d'avancer l'élément courant, l'adresse de l'élément précédent est sauvegardée. Pour réaliser cette opération, il y a trois actions à exécuter dans cet ordre (voir l'algorithme I.5) :

- Créer le nouvel élément.
- Localiser l'élément précédent.
- Relier l'élément précédent avec le nouvel élément.

Exemple :
algorithme Ajout_milieu ;
type liste =^cellule ;
cellule = enregistement
info : entier ;

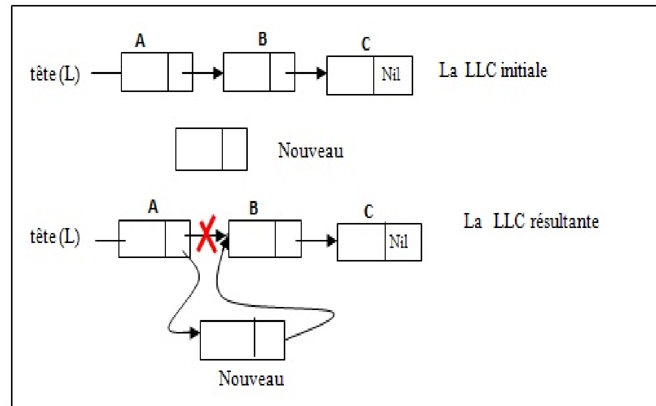


FIGURE 1.4 – L'insertion d'un nouvel élément au milieu d'une LLC.

suivant : liste ;
fin ;
var L, precedent : liste ;
debut
| Allouer(nouveau) ; */* Réserve un espace mémoire pour le nouveau élément*/*
| Lire (*nouveau*^.info) ; */* Stocke dans l'Info de l'élément pointé par nouveau la valeur saisie*/*
| L ← precedent ; */* Mémoriser la tête de L dans une autre liste pour ne pas la perdre */*
| **TQ**(la condition d'arrêt n'est pas vérifiée) **faire** */* Localiser l'endroit d'insertion*/*
| | precedent ← *precedent*^.suivant ; */* Passer d'un élément à l'autre*/*
| | **finTQ ;**
| *nouveau*^.suivant ← *precedent*^.suivant ; */* Relier le nouveau élément avec son suivant */*
| *precedent*^.suivant ← nouveau ; */* Relier le nouveau élément avec son précédent */*
fin.

Algorithme I.5. Insérer une nouvelle cellule au milieu d'une LLC.

1.6.3.3 Insertion d'un élément à la fin d'une LLC

Pour réaliser cette opération, il y a deux actionsà exécuter dans cet ordre (voir l'algorithme I. 6) :

- Créer le nouvel élément dont sa partie suivant est Nil.
- Récupérer l'adresse du dernier élément.
- Relier le dernier élément avec le nouvel élément de telle sorte que le nouvel élément devienne le suivant du dernier élément.

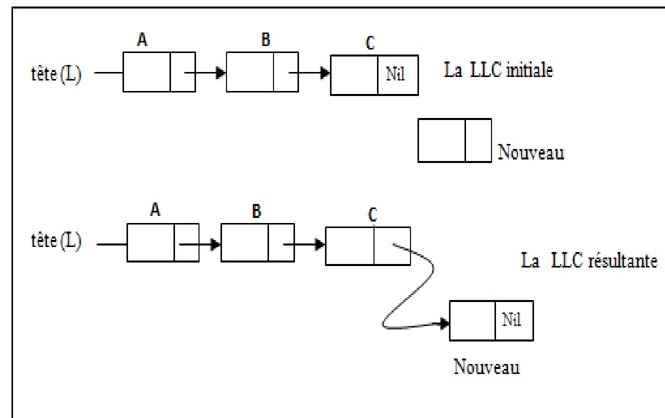


FIGURE 1.5 – L'insertion d'un nouvel élément à la fin d'une LLC.

Exemple :

```
algorithme Ajout_fin;
type liste =^cellule
        cellule = enregistement
            info : entier;
            suivant : liste;
        fin;
var L, nouveau, P : liste;
debut
|     Allouer(nouveau );          /* Réserver un espace mémoire pour le nouveau élément*/
|     Lire (nouveau^.info);       /* Stocker dans l'Info de l'élément pointé par nouveau la valeur saisie*/
|     nouveau^.suivant ← Nil;     /* Mettre Nil dans le suivant de nouveau, car, il sera le dernier élément de L. */
|     P ← L;    /* Mémoriser la tête de L dans une autre liste pour ne pas la perdre */
|     TQ( P^.suivant < > Nil) faire /* Parcourir L jusqu'à le dernier élément*/
|     |    P ← P^.suivant; /* Passer d'un élément à l'autre*/
|     | finTQ;
|     P^.suivant ← nouveau;       /* Relier le dernier élément de L avec nouveau */
fin.
```

Algorithme I.6. Insérer une nouvelle cellule à la fin d'une LLC.

1.6.4 Suppression d'un élément d'une LLC

Pour supprimer un élément de la liste, il faut le repérer en premier lieu et le détruire s'il existe tout en gardant le chaînage de la liste. Comme l'opération de l'insertion, trois cas sont possibles :

- La suppression du premier élément de la LLC,
- La suppression d'un élément se trouvant au milieu de la LLC,
- La suppression du dernier élément de la LLC.

1.6.4.1 Suppression du premier élément d'une LLC

La suppression du premier élément suppose de bien actualiser la valeur du pointeur premier qui indique toujours le début de la liste. Pour réaliser cette opération, il y a deux actions à exécuter dans cet ordre :

- faire pointer la tête de liste sur le deuxième élément de la liste,
- libérer l'espace mémoire occupé par l'élément supprimé.

Il est nécessaire de déclarer un pointeur P qui va pointer sur l'élément à supprimer et permettre de libérer l'espace qu'il occupait.

Exemple :

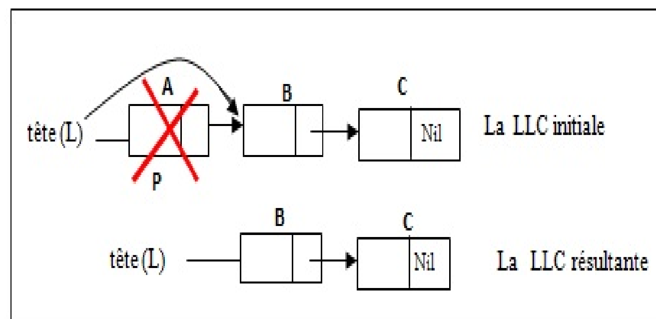


FIGURE 1.6 – Suppression du premier élément d'une LLC.

```
algorithme supprimer_premier ;
type liste =^cellule ;
        cellule = enregistement
            info : entier ;
            suivant : liste ;
        fin ;
var L, P : liste ;
debut
|    si L < > Nil alors /* La liste n'est pas vide on peut donc supprimer le premier élément*/
|     | P ← L ; /* P pointe sur le premier élément de L*/
|     | L ← L^.suivant ; /* L pointe sur le deuxième élément */
|     | Libérer (P) ; /* Libération de l'espace mémoire qu'occupait le premier élément */
|     sinon
|     | Ecrire ("la liste L est vide") ;
```

| **finsi ;**
fin.

Algorithme I.7. Supprimer le premier élément d'une LLC.

1.6.4.2 Suppression d'un élément se trouvant au milieu d'une LLC

L'objectif est de supprimer un élément quelconque de la LLC en fonction d'un critère donné. Le critère ici est que le champ *info* de l'élément soit égal à une valeur donnée. La recherche s'arrête si un élément répond au critère et nous supprimons uniquement le premier élément trouvé s'il y en a un. Comme pour la suppression précédente, la LLC ne doit pas être vide. Pour réaliser cette opération, il faut :

- Trouver l'adresse **P** de l'élément à supprimer,
- Sauvegarder l'adresse **Precedent** de l'élément précédant l'élément pointé par **P** pour connaître l'adresse de l'élément précédant l'élément à supprimer, puis faire pointer l'élément précédent sur l'élément suivant l'élément à supprimer,
- Libérer l'espace mémoire occupé par l'élément supprimé.

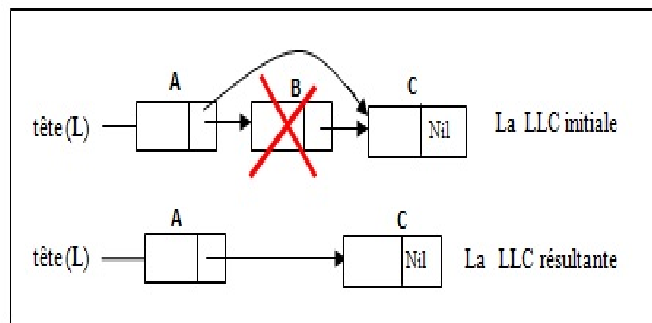


FIGURE 1.7 – Suppression d'un élément qui se trouve au milieu d'une LLC.

Exemple :
algorithme supprimer_milieu ;
type liste =^cellule
    cellule = enregistement
        info : entier ;
        suivant : liste ;
    fin ;
var L, P, precedent : liste ;
debut

```
|    si L < > Nil alors /* La liste n'est pas vide on peut donc supprimer le premier élément */
|     | Precedent ← L ; /*Pointeur précédent */
|     | P ← L^.suivant ; /* Pointeur courant */
|    | TQ( l'élément à supprimer n'est pas encore trouvé) faire /* Localisation du l'élément à supprimer*/
|     |     | Precedent← P ; /*On garde la position du précédent */
|     |     | P← P^.suivant ; /* On passe à l'élément suivant dans la liste */
|     | finTQ ;
|     | Precedent^.suivant ← P^.suivant ; /*On "saute" l'élément à supprimer */
|     | Libérer (P) ; /*Libération de l'espace mémoire qu'occupait l'élément P*/
|    sinon
|     | Ecrire ("la liste L est vide") ;
|     | finsi ;
fin.
```

Algorithme I.8. Supprimer l'élément qui se trouve au milieu d'une LLC.

1.6.4.3 Suppression d'un élément se trouvant à la fin d'une LLC

La suppression du dernier élément consiste à changer le pointeur de l'avant dernier élément à Nil, ce qui le rend le nouveau dernier élément, puis de libérer l'espace occupé par le dernier élément. Les différentes étapes, permettant de réaliser cette opération sont :

- Localiser le dernier élément ainsi que l'élément qui le précède,
- Libérer l'espace occupé par le dernier élément,
- Mettre Nil dans la partie suivant de l'élément avant le dernier.

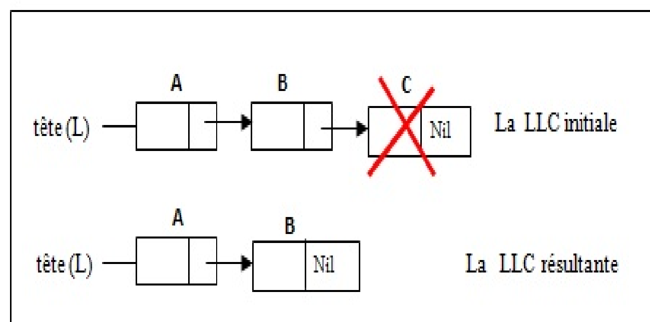


FIGURE 1.8 – Suppression du dernier élément d'une LLC.

Exemple :

```
algorithme supprimer_dernier ;
type liste =^cellule
```

```
    cellule = enregistement
        info : entier;
        suivant : liste;
    fin;
var L, P, precedent : liste;
debut
|   si L < > Nil alors /* La liste n'est pas vide on peut donc supprimer le premier élément */
|     | Precedent ← L; /*Pointeur précédent */
|     | P ← L^.suivant; /* Pointeur courant */
|     | TQ(P^.suivant < > Nil) faire /* Localisation du dernier élément*/
|     |     | Precedent← P; /*On garde la position du précédent */
|     |     | P← P^.suivant; /* On passe à l'élément suivant dans la liste */
|     | finTQ;
|     | Libérer (P); /*Libération de l'espace mémoire qu'occupait l'élément P*/
|     | Precedent^.suivant ← Nil; /* Mettre Nil dans la partie suivant du precedent */
|   sinon
|     | Ecrire ("la liste L est vide");
|     | finsi;
fin.
```

Algorithme I.9. Supprimer le dernier élément d'une LLC.

1.7 Exercice

Soit l'algorithme qui assure la gestion des logements à louer. Pour chaque logement, implémenté avec une LLC, on retient son identificateur (entier), sa catégorie (économique, social, ou normal), son prix (réel) et sa localité (chaine [50]). On s'intéresse aux fonctionnalités suivantes :

1. Créer une liste chainée L permettant de représenter l'ensemble des logements.
2. Afficher les informations de tout logement dont la catégorie est "économique".
3. Supprimer la cellule ayant l'identificateur *ident* de la liste L. L'identificateur *ident* doit être lu à partir du clavier.
4. Créer trois sous listes Léconomique, Lsocial, et Lnormal (à partir du champ catégorie) à partir de la liste L et sans allocation de nouveaux espaces.
5. On suppose que les trois sous listes créées Léconomique, Lsocial, et Lnormal sont triées suivant l'identificateur de logement. Ajouter un logement dont les informations sont introduites par clavier.

Remarque : Vous pouvez utiliser des sous programmes.

Chapitre 2

Les piles

2.1 Introduction

La structure de données *Pile* est une structure liste particulière, dans laquelle les opérations d'insertion et de suppression s'effectuent à une seule extrémité de la liste.
Dans ce chapitre, nous présenterons en langage algorithmique les principales opérations qui peuvent être appliquées sur cette structure. Nous monterons, également, l'importance de cette structure à travers quelques exemples d'applications.

2.2 Définition

Une pile est une structure de données qui stocke de manière ordonnée des éléments mais rend accessible uniquement un seul élément d'entre eux, appelé le sommet de la pile. Quand on ajoute un élément, celui-ci devient le sommet de la pile, c'est-à-dire le seul élément accessible. Quant on retire un élément de la pile, on retire toujours le sommet, et le dernier élément ajouté avant lui devient alors le sommet de la pile. Pour résumer, le dernier élément ajouté dans la pile est le premier élément à en être retiré. Cette structure est également appelée une liste LIFO (Last In, First Out).

2.3 Implémentation des piles

Généralement, il y a deux façons pour représenter une pile. La première peut se faire à l'aide d'un tableau (elle sera traitée dans la séance de TD). La seconde s'appuie sur la structure de LLC vue précédemment (voir le chapitre I).

2.3.1 Implémentation par des tableaux

L'implémentation statique des piles utilise les tableaux. Dans ce cas, la capacité de la pile est limitée par la taille du tableau. L'ajout à la pile se fait dans le sens croissant des indices, tandis que le retrait se fait dans le sens inverse.

2.3.2 Implémentation par des LLCs

L'implémentation dynamique utilise les LLC. Dans ce cas, la pile peut être vide mais ne peut être jamais pleine, sauf bien sûr en cas d'insuffisance de l'espace mémoire (dans notre cours, on considérera que cette implémentation). Le sommet de la pile est le premier élément de la pile et le pointeur de tête pointe sur ce sommet. D'abord, il faut commencer par définir un type de variable pour chaque élément de la pile. La déclaration est identique à celle d'une liste chaînée, par exemple pour une pile de valeurs entières, on aura la déclaration suivante :

```
type Pile =^Element ;
     Element = enregistement
        info : entier ;
        suivant : Pile ;
     fin ;
```

Une fois le type Pile est défini, on peut déclarer une variable de ce type de la manière suivante :

```
var P : Pile ;
```

2.4 Représentation graphique d'une pile

Avec la structure de données Pile, les éléments sont ajoutés un par un selon leur ordre d'arrivée et ils sont retirés un par un en partant du dernier arrivé appelé le sommet de la pile (voir la figure II.1). Avec cette structure, on ne travaille que sur le sommet de la pile, en

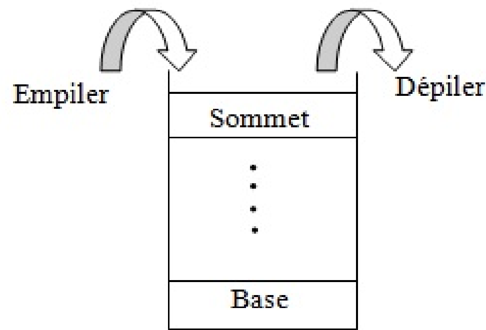


FIGURE 2.1 – Représentation graphique d'une Pile.

l'associant les deux termes suivants :
- Empiler pour ajouter un élément,
- Dépiler pour supprimer un élément.

2.5 Opérations sur les piles

Les opérations autorisées sur une pile sont :

- Pile vide ?, pour vérifier si la pile est vide ou non,
- Empiler, toujours au sommet, et jusqu'à la limite de la mémoire,
- Sommet de pile, pour retourner le premier élément de la pile,
- Dépiler, toujours au sommet, si la pile n'est pas vide,
- Vider une pile.

Avec l'implémentation dynamique en utilisant les LLC, le sommet de la pile correspondant à la tête de la liste. Dans ce cas, le test à vide correspond au test à vide des listes chaînées, l'empilement correspond à l'insertion en tête, la lecture du sommet de la pile correspond à la lecture de l'élément de tête de la liste et le dépilement à une combinaison de lecture de l'élément de tête et de suppression.

2.5.1 Pile vide ?

Une pile est vide signifie que le pointeur de tête est mis à Nil. Pour le confirmer, il suffit de faire un test sur la valeur du pointeur de tête. La fonction **pile_vide** retourne le résultat du test sur la valeur du pointeur de tête, premier de la liste, **vrai** si oui, **faux** sinon :
fonction pile_vide (P : Pile) : booleen ;

debut
| **si**(P = Nil) **alors** */* La pile est vide */*
| | pile_vide ← vrai ;
| **sinon** */* La pile contient au moins un élément */*
| | pile_vide ← faux ;
| | **finsi ;**
fin ;

2.5.2 Empiler un élément sur une pile

Cette fonction empile l'élément **e** au sommet de la pile **P**. Cela revient à ajouter l'élément **e** en tête de la liste.
procedure empiler (**var** P : Pile ; e :entier) ;
var Q : Pile ;
debut
| Allouer (Q) ; */* Réserver un espace mémoire pour le nouvel élément*/*
| Q^.info ← e ; */*stocker dans l'Info de l'élément la valeur e passée en paramètre*/*
| Q^.suivant ← P ; */*stocker dans Suivant du nouvel élément l'adresse de P*/*
| P ← Q ; /* *tête pointe maintenant sur le nouvel élément */*
fin ;

2.5.3 Sommet de la pile

Le sommet de la pile est donné en permanence par le pointeur tête (adresse de la liste). Pour lire le sommet, il suffit de considérer les valeurs du premier élément à cette adresse. Si on suppose que P est une pile des entiers, la fonction **Sommet** sera comme suit :
fonction Sommet (P : Pile) : entier ;
debut
| **si** (non (pile_vide (P))) **alors** */* La pile n'est vide */*
| | Sommet ← P^.info ; */* récupérer la valeur stockée dans le sommet de la pile */*
| **sinon**
| | Ecrire ("la pile est vide") ;
| | **finsi ;**
fin ;

2.5.4 Dépiler un élément d'une pile

Dépiler c'est supprimer le premier élément de la liste. Le premier élément de la liste passe alors au suivant et l'espace occupé par l'élément à supprimer sera libéré :

```
procedure depiler ( var P : Pile) ;
var Q : Pile ;
debut
|    si (non (pile_vide (P))) alors /* La pile n'est vide */
|     | Q← P ; */Q pointe sur le Sommet de la pile*/
|     | P ← P^.suivant ; /*Le deuxième élément devient le nouveau Sommet*/
|     | Liberer (Q) ; /*Libérer l'espace occupé par l'ancien Sommet*/
|     sinon
|     | Ecrire ("la pile est vide") ;
|     | finsi ;
fin ;
```

2.5.5 Vider une pile

Vider et détruire sont identiques dans une pile dynamique. Il s'agit de dépiler un à un tous les éléments de la pile et de libérer la mémoire qu'ils occupent. A la fin, il est important de bien initialiser le pointeur de tête à Nil :

```
procedure vider_pile ( var P : Pile) ;
var Q : Pile ;
debut
|    TQ (non (pile_vide (P))) faire /* Tant que la pile n'est pas vide, retirer le Sommet et le supprimer */
|     | Q← P ; */Q pointe sur le Sommet de la pile*/
|     | P ← P^.suivant ; /* Passer à l'élément suivant*/
|     | Liberer (Q) ; /*Libérer l'espace occupé par l'ancien Sommet*/
|     finTQ ;
|     P ← Nil ;
fin ;
```

2.6 Applications importantes des piles

En informatique, les piles sont utilisées pour implémenter les appels des sous programmes, en particulier lors des appels récursifs. Elles sont utilisées également dans l'évaluation des

expressions arithmétiques, etc.

2.6.1 Les appels récursifs

Un algorithme récursif est un algorithme qui contient une ou plusieurs actions paramétrées récursives. Une action paramétrée récursive est le fait que cette action paramétrée peut s'appeler elle même avant que sa première exécution ne soit terminée. En d'autres termes, une action paramétrée **P** est dite récursive si son exécution provoque un ou plusieurs appels de **P**. Ces appels sont dits récursifs. Pour la résolution d'un problème **PB**, **l'écriture d'une action paramétrée récursive** est fondée sur les deux points suivants :

1. La résolution du **PB** dans certains cas particuliers ou triviaux (la condition d'arrêt),
2. Le choix d'une décomposition du **PB** en sous problèmes de même nature que **PB** et de telle sorte que les décompositions successives aboutissent toujours l'un des cas particuliers cités en 1.

Exemple :
L'algorithme récursif réalisé par la fonction **Fact** qui calcule la factorielle d'un entier N.

```
fonction Fact (N : entier) : entier;
debut
|    si (N = 1) alors /* La condition d'arrêt */
|      | Fact← 1;
|      sinon
|      | Fact ← N*Fact(N-1); /* L'appel récursif*/
|      finsi;
fin;
```

Le résultat de l'évaluation de 4 !
$Fact(4) = 4*Fact(3) = 4*3*Fact(2) = 4*3*2*Fact(1) = 4*3*2*1 = 4*3*2 = 4*6 = 24.$
Le schéma suivant montre l'évolution du contenu de la pile en exécutant cette fonction pour N= 4.

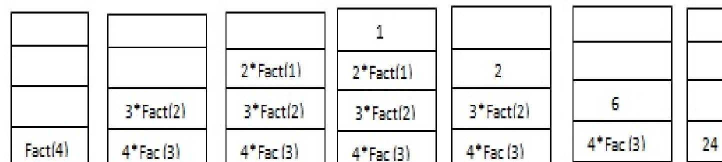


FIGURE 2.2 – Schéma représentant l'utilisation de la pile pour calculer la factorielle d'un entier.

2.6.2 L'évaluation d'une expression arithmétique

Une application courante des piles se fait dans le calcul arithmétique, l'ordre dans la pile permet d'éviter l'usage des parenthèses.
La notation usuelle, comme $(3+5)*2$, est dite infixée. Son défaut est qu'elle nécessite l'utilisation de parenthèses pour éviter toute ambiguïté (ici, avec $3+(5*2)$).
Pour éviter le parenthésage, il est possible de transformer une expression infixée en une expression postfixée en faisant "glisser" les opérateurs arithmétiques à la suite des expressions auxquelles ils s'appliquent.
Exemple :
L'expression $((a+(b*c))/(cd)$ est exprimée, en postfixé, comme suit : $bc*a+cd-/$. Pour l'évaluer, on utilise une pile. On parcourt l'expression de gauche à droite, en exécutant l'algorithme suivant :

```
debut
|    i ← 1;
|   TQ (i < Longueur (Expression))faire
|     | si( Expression[i] est un Opérateur) alors
|     |      | Retirer deux éléments de la pile;
|     |      | Calculer le résultat selon l'opérateur;
|     |      | Mettre le résultat dans la pile;
|     | sinon
|     |      | Mettre l'opérande dans la pile;
|     | finsi;
|     finTQ;
fin.
```

Algorithme II.1. L'évaluation d'une expression postfixée par une pile.

Le schéma suivant montre l'évolution du contenu de la pile en exécutant cet algorithme sur l'expression précédente.

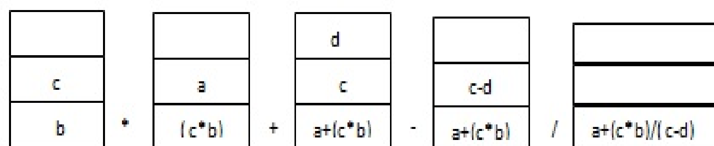


Figure 2.3 – Schéma représentant l'évaluation d'une expression postfixée par une pile.

2.7 Exercice

Soit une pile de nombres entiers, ordonnées suivant l'ordre décroissant des valeurs. Ecrire un algorithme qui insère, dans cette pile, une valeur VAL donnée par l'utilisateur, si elle n'existe pas.

Chapitre 3

Les files

3.1 Introduction

Il existe différents type de listes linéaires. Un exemple tiré de la vie courante peut être le rangement de vêtements et son organisation comportementale. Dans certains types de listes les opérations d'insertion et de suppression se déroulent toujours aux extrémités de la liste. Ce type de listes se rencontrant très fréquemment est appellé **File**.

3.2 Définition

La file est une liste qui stocke les éléments avec un ordre spécifique. C'est comme une file d'attente à la caisse d'un magasin. Les personnes arrivent une à une et constituent la file et elles partent une à une dans l'ordre d'arrivée après avoir payé. Donc, une file est définie comme une collection d'éléments dans laquelle tout nouveau élément est inséré à la fin et tout élément ne peut être supprimé que du début. C'est l'ordre du premier arrivé premier sorti, "First In First Out" dit FIFO en informatique.

3.3 Implémentation d'une file

On peut implémenter une file à l'aide d'un tableau (file statique) ou par une liste chaînée (file dynamique). C'est l'implémentation en liste chaînée qui est retenue dans ce qui suit. Il est nécessaire d'avoir deux pointeurs pour une file. Un pointeur pour gérer les entrées en **queue**

de liste et un autre pour gérer les sorties en **tête** de liste. En dynamique, une file n'a pas de taille limite, hormis celle de la mémoire centrale (RAM) disponible. Il faut commencer par définir un type de variable pour chaque élément de la file. La déclaration est identique à celle d'une liste chaînée. Par exemple, pour une file des entiers :

```
type File =^Element;
     Element = enregistement
        info : entier;
        suivant : ^File;
     fin;
```

Une fois le type File est défini, on peut déclarer des variables de ce type de la manière suivante :
var Tete, Queue : File;

3.4 Représentation graphique d'une file

Une file est une liste dont nous conservons en permanence le **premier** et le **dernier** élément de la file. Une file est ainsi définie par deux pointeurs, celui de tête pour les sorties et celui de queue pour les entrées. A part cette spécificité, c'est un cas normal des LLC.
On associe, à la structure File, les termes suivants :

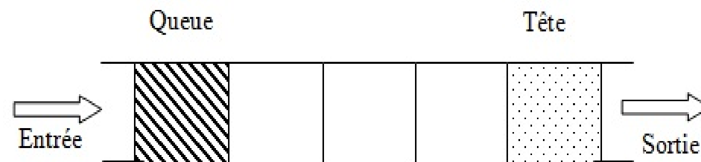


FIGURE 3.1 – Représentation graphique d'une file.

Premier : représente l'élément qui se trouve en tête de la file,
Dernier : représente l'élément qui se trouve en queue de la file,
Enfiler : pour ajouter un élément à la file, et qu'il sera le dernier,
Défiler : pour supprimer le premier élément.

3.5 Opérations sur les files

Les opérations autorisées sur une file sont :
- Vérifier si la file est vide ou non,
- Premier de la file, pour retourner le premier élément de la file,

– Dernier de la file, pour retourner le dernier élément de la file,
– Enfiler, toujours, à la queue et jusqu'à la limite de la mémoire,
– Défiler, toujours, de la tête si la file n'est pas vide,
– Vider une file.

3.5.1 File vide ?

Pour savoir si une file est vide, il suffit de tester les pointeurs de Tete ou de Queue, si c'est à Nil alors la file est vide. Notons qu'il n'est pas possible que l'un des deux soit à Nil et l'autre non :

```
fonction file_vide ( F : File ) : bouleen;
debut
|    si( Tete = Nil) alors /* La file est vide */
|     | file_vide ← vrai;
|     sinon /* Il y a au moins un élément */
|     | file_vide ← faux;
|     | finsi;
fin;
```

3.5.2 Le premier et le dernier élément de la file

Récupérer les données du premier ou du dernier élément de la file consiste à récupérer les adresses des pointeurs Tete et Queue, sous réserve que la file ne soit pas vide.

```
procedure Premier_File ( F : File; var E :entier);
debut
|    si (non (file_vide (F))) alors /* La file n'est vide */
|     | E ← Tete^.info; /* récupérer la valeur du premier élément dans la file */
|     sinon
|     | Ecrire ("la file est vide");
|     | finsi;
fin;
```

```
procedure Dernier_File ( F : File; var E :entier);
debut
|    si (non (file_vide (F))) alors /* La file n'est vide */
|     | E ← Queue^.info; /* récupérer la valeur du denier élémént dans la file */
```

| **sinon**
| | Ecrire ("la file est vide") ;
| | **finsi ;**
fin ;

3.5.3 Enfiler un élément dans une file

Enfiler un élément dans une file consiste à ajouter un nouvel arrivant à la file. L'ajout se fait en queue de la file. Si la file est vide la queue et la tête prennent la même valeur, celle du nouvel élément, ce qui donne :
procedure Enfiler (**var** F : File ; val :entier) ;
var P : File ;
debut
| Allouer (P) ; */* Réserver un espace mémoire pour le nouvel élément*/*
| P^.info ← val ; */*stocker dans l'Info de l'élément la valeur val passée en paramètre*/*
| **si** (file_vide (F)) **alors** */* La file est vide */*
| | P^.suivant ; ← Nil ; */* P est l'élément unique, mettre donc Nil dans son suivant */*
| | Tete ← P ; */* Tête pointe maintenant sur l'élément unique */*
| **sinon** */* il y a au moins un élément */*
| | P^.suivant ← Queue ; */* Le nouvel élément est ajouté au début de la file */*
| | **finsi ;**
| Queue ← P ; */* Queue pointe sur P*/*
fin ;

3.5.4 Défiler un élément d'une file

Défiler est équivalent à dépiler et consiste à supprimer l'élément de tête si la file n'est pas vide. Il faut conserver l'adresse de l'élément qu'on supprime pour libérer sa place. Deux cas sont à envisager :

- Si un seul élément dans la liste retourner son adresse et mettre les deux pointeurs Tete et Queue à Nil.
- Si plusieurs, retourner l'adresse du premier et le premier prend l'adresse de l'élément qui le précède. Ce qui donne :

procedure Defiler (**var** F : File) ;
var P, Precedent : File ;
debut
| **si** (non (File_vide (F)) **alors** */* La file n'est pas vide */*

```
|    | si(Queue = Tete) alors /* la file contient un seul élément */
|    |    | P← Queue ; /*P pointe sur l'élément unique */
|    |    | Tete← Nil ;
|    |    | Queue ← Nil ;
|    |    | Liberer (P) ; /*Libération de l'espace mémoire qu'occupait P */
|    | sinon ;/* La file contient plusieurs éléments, il faut localiser le premier élément pour le supprimer*/
|    |    | Precedent← Queue ; /*Precedent devient le nouveau premier élément dans la file */
|    |    | P← Queue^.suivant ; /* Mettre Nil dans la partie suivant de la nouvelle Tete */
|    |    | TQ (P < > Tete) faire /*Localisation du premier élément*/
|    |    |    | Precedent← P ; /* On garde la position du précédent */
|    |    |    | P ← P^.suivant ; /* on passe à l'élément suivant dans la file*/
|    |    | finTQ ;
|    |    | Tete← Precedent ; /*Precedent devient le nouveau premier élément dans la file */
|    |    | Tete^.suivant← Nil ; /* Mettre Nil dans la partie suivant de la nouvelle Tete */
|    |    | Liberer (P) ; /*Libération de l'espace mémoire qu'occupait l'ancienne Tete */
|    | finsi ;
|    sinon
|    | Ecrire ("la file F est vide") ;
|    | finsi ;
fin ;
```

3.5.5 Vider une file

Vider la file consiste à défiler tous les éléments jusqu'à ce que la file soit vide et à chaque fois en libérant l'espace mémoire alloué pour l'élément, ça donne :

```
procedure Vider_File ( var F : File) ;
var P : File ;
debut
|    TQ (non (file_vide (F))) faire /* Tant que la file n'est pas vide, retirer le dernier élément et le supprimer */
|    | P← Queue ; */Q pointe sur le dernier*/
|    | Queue ← Queue^.suivant ; /* Passer à l'élément suivant*/
|    | Liberer (P) ; /*Libérer l'espace occupé par P*/
|    finTQ ;
|    Tete ← Nil ;
|    Queue ← Nil ;
fin ;
```

3.6 Applications importantes des files

L'application la plus classique est la file d'attente, et elle sert beaucoup en simulation. Elle est, aussi, très utilisée aussi bien dans la vie courante que dans les systèmes informatiques. Par exemple, elle modélise la file d'attente des clients devant un guichet, les travaux en attente d'exécution dans un système de traitement par lots, ou encore les messages en attente dans un commutateur de réseau téléphonique. On retrouve également les files d'attente dans les programmes de traitement de transactions telle que les réservations de sièges d'avion ou de billets de théâtre. Les applications de la file en informatique sont nombreuses, nous citons :

- **Impression de programmes**
  Maintenir une file de programmes en attente d'impression.
- **Disque driver**
  Maintenir une file de requêtes disque d'input/output disque.
- **Ordonnanceur (dans les systèmes d'exploitation)**
  Maintenir une file de processus en attente d'un temps machine.

3.7 Exercice

Reprendre l'exercice donné dans le chapitre précédent (Chapitre II : Les piles), en considérant la structure de données File à la place de Pile.

Chapitre 4

Les arbres

4.1 Introduction

Les structures **arborescentes** sont des structures de données parmi les plus importantes en informatique. Plus générales que les listes, elles servent à représenter des ensembles dont les éléments sont ordonnées par une relation d'ordre noté $<$ (au sens strict). Elles sont, aussi, appelées structures de données **hiérarchiques**.

4.2 Définition

Un arbre est une **SD non linéaire** (chaque élément de l'arbre, bien que n'ayant qu'un prédécesseur, peut avoir plus d'un successeur). C'est une structure SD qui généralise la liste, alors qu'une cellule de liste possède un seul successeur (son suivant), dans un arbre il peut y en avoir plusieurs. On parle, alors, de nœud (au lieu de cellule). Un nœud **père** peut avoir plusieurs nœuds fils. Un **fils** n'a qu'un seul père et tous les nœuds ont un ancêtre commun appelé la **racine** de l'arbre (le seul nœud qui n'a pas de père).

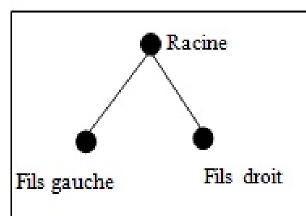


Figure 4.1 – Schéma représentant la notion d'un arbre.

4.3 Représentation graphique d'un arbre

Pour pouvoir repérer les nœuds dans un arbre, il faut associer à chacun d'entre eux un nom différent. Pour schématiser un arbre, on symbolise souvent un nœud par un nom (nombre ou chaine de caractères) inscrit dans un **cercle**. Les nœuds sont liés par une relation de **parenté**, symbolisée par un **arc** (voir le schéma suivant).

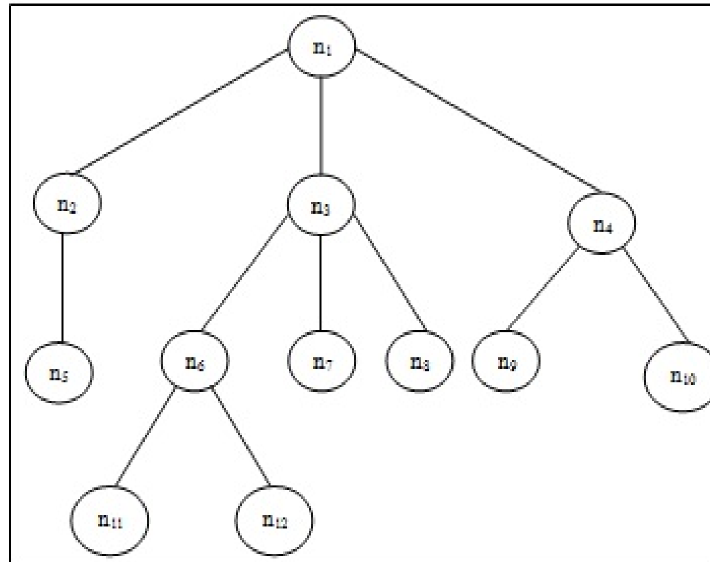


FIGURE 4.2 – Représentation graphique d'un arbre.

- **Terminologie informatique des arbres**
  On associe à un arbre, généralement, les termes suivant :
  - **Frère** : deux nœuds qui ont le même père sont des frères. Dans l'exemple ci-dessus, les nœuds n_6, n_7 et n_8 sont des frères.
  - **Chemin** : une séquence de nœuds partant d'un nœud n_i en suivant des arcs jusqu'à un nœud n_j s'appelle un chemin de n_i à n_j. Dans l'exemple ci-dessus, le chemin depuis la racine n_1 jusqu'au nœud n_{12} est le suivant : $(n_1, n_3), (n_3, n_6), (n_6, n_{12})$.
  - **Ascendant ou ancêtre, descendant** : on dit qu'un nœud n_i est un ascendant ou ancêtre d'un nœud n_j , s'il existe un chemin entre n_i et n_j (n_j est dit descendant de n_i). Dans l'exemple ci-dessus, n_3 est un ancêtre de n_{11} et n_{11} lui-même représente un descendant de n_3.
  - **Nœud externes ou feuill**e : un nœud qui n'a pas de fils est appelé nœud externe ou feuille. Dans l'exemple ci-dessus, les nœuds n_5, n_7, n_8, n_9, n_{10}, n_{11}, n_{12} sont des feuilles.
  - **Nœud interne** : un nœud qui a deux fils (ou plus) est appelé nœud interne ou point double, par contre un nœud qui a seulement un fils gauche (resp. droit) est appelé point simple à gauche (resp. à droite) ou nœud interne au sens large. Dans l'exemple

ci-dessus, n_3, n_4 et n_6 sont des nœuds internes, par contre, n_2 est un nœud interne au sens large.

- **Branche** : on appelle branche d'un arbre A tout chemin de la racine à une feuille de A. Dans l'exemple ci-dessus, les arcs suivants $(n_1, n_3), (n_3, n_6)$ et (n_6, n_{11}) constituent une branche.
- **Bord** : on appelle bord gauche (resp. droit) d'un arbre A, le chemin obtenu à partir de la racine en ne suivant que des liens gauches (resp. droits). Dans l'exemple ci-dessus, les arcs suivant (n_1, n_2) et (n_2, n_5) représentent le bord gauche. Alors que, les arcs (n_1, n_4) et (n_4, n_{10}) représentent le bord droit.
- **Arbre ordonné** : un arbre ordonné est un arbre dans lequel l'ensemble des fils de chaque nœud est totalement ordonné.
- **Sous-arbre** : un sous-arbre d'un arbre est l'ensemble de n'importe quel nœud de l'arbre et tous ses descendants. Le sous-arbre du nœud n est le sous-arbre dont la racine est un fils de n. Dans l'exemple ci-dessus, le sous-arbre possédant n_3 comme racine est un sous-arbre du nœud n_1.
- **Forêt** : une forêt est un ensemble fini d'arbres.

• **Mesures sur les arbres**

On introduit, dans ce qui suit, quelques mesures sur les arbres à valeurs dans les entiers ou réels :

- **Taille d'un arbre** : c'est le nombre total des nœuds de cet arbre. Dans l'exemple ci-dessus, la taille de l'arbre $= 12$.
- **Degré d'un nœud** : c'est le nombre de fils de ce nœud. Dans l'exemple ci-dessus, le degré du nœud $n_4 = 2$. Par contre, le degré du nœud $n_3 = 3$.
- **Degré d'un arbre** : c'est le degré maximum atteint par les nœuds de cet arbre. Dans l'exemple ci-dessus, le degré de l'arbre est = degré maximal de ses nœuds = degré du nœud $n_3 = 3$.
- **Hauteur/ Profondeur ou Niveau d'un nœud** : représente le nombre d'arêtes entre le nœud et la racine. Dans l'exemple ci-dessus, la hauteur de la racine $n_1 = 0$. La hauteur du nœud $n_2 =$ la hauteur du nœud $n_3 =$ la hauteur du nœud $n_4 = 1$. La hauteur du nœud $n_{11} =$ la hauteur du nœud $n_{12} = 3$. Mathématiquement, on peut l'exprimer ainsi :

  Soit A un arbre et x étant un nœud de A. La hauteur H du nœud x est :

$$H(x) = \begin{cases} 0, & \text{si } x \text{ est la racine de } A; \\ 1 + H(y), & \text{si } y \text{ est le père de } x; \end{cases} \tag{4.1}$$

- **Hauteur / Profondeur d'un arbre** : représente la hauteur maximale de ses nœuds. Soit A un arbre, la hauteur **H** de l'arbre A est donnée par :
  $H(A) = max\{H(x),\ x$ nœud de A $\}$. Dans l'exemple ci-dessus, la hauteur de l'arbre $= 3$.
- **Longueur de cheminement externe d'un arbre** :

Soit A un arbre, la longueur de cheminement externe de cet arbre **LCE (A)** est donnée par :
$LCE(A) = \sum H(f)$, tel que f représente une feuille dans A. Dans l'exemple ci-dessus, LCE = H (n5) + H (n11) + H (n12) + H (n7) + H (n8) + H (n9) + H (n10) = 2+3+3+2+2+2+2 = 16.

- **Longueur de cheminement interne d'un arbre** :
  Soit A un arbre, la longueur de cheminement interne de cet arbre **LCI** (A) est donnée par :
  $LCI(A) = \sum H(x)$, tel que x représente un nœud interne ou un un nœud interne au sens large de A. Dans l'exemple ci-dessus, LCI = H (n2) + H (n3) + H (n4) + H (n6) = 1+1+1+2 = 5.
- **Longueur de cheminement d'un arbre** :
  Soit A un arbre, la longueur de cheminement de cet arbre **LC** (A) est donnée par :
  LC(A) = LCE(A) + LCI(A). Dans l'exemple ci-dessus, LC = 16 + 5 = 21.

4.4 Arbre binaires

Selon le nombre des nœuds, il existe deux types d'arbres : Arbre n-aire et Arbre binaire.

4.4.1 Notion d'un arbre n-aire

Les arbres n-aires ou les arbres généraux sont des arbres dont chaque nœud peut accepter de *0* à n nœuds. Ces arbres sont en général peu utilisés dans les algorithmes parce qu'ils peuvent toujours être convertis en arbres binaires qui sont plus simples à manipuler.

4.4.2 Notion d'un arbre binaire

Les arbres binaires sont une restriction des arbres généraux. Au lieu de comporter un nombre indéterminé de fils, on fixe une limite de deux fils par nœud : un fils gauche et un fils droit. Autrement dit, un arbre binaire est soit vide (noté $\emptyset$) soit de la forme $B =< 0, B_1, B_2 >$ où B_1 et B_2 sont des arbres binaires disjoints et *0* est un nœud appelé *racine*.
Les arbres binaires sont majoritairement utilisés dans les algorithmes. Pour cette raison, on retient seulement ce type d'arbres dans le reste de ce chapitre.

4.4.3 Arbres binaires particuliers

Dans ce type d'arbres, on peut trouver des arbres particuliers comme :

4.4.3.1 Arbre binaire complet

Un arbre binaire est complet si toutes ses branches ont la même longueur et tous ses nœuds qui ne sont pas des feuilles ont deux fils. Soit A un arbre binaire complet. Le nombre de nœuds de A au niveau *0* est *1*, le nombre de nœuds au niveau *1* est *2*,.. , et le nombre de nœuds au niveau p est donc 2^p . En particulier le nombre de feuilles est donc 2^{h-1} si h est le nombre de niveaux.

Exemple

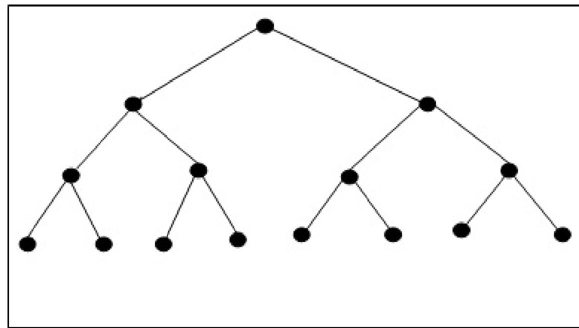


FIGURE 4.3 – Exemple d'un arbre binaire complet.

4.4.3.2 Arbre binaire localement complet

Un arbre binaire localement complet est un arbre binaire dont chacun des nœuds possèdent soit *0* soit *2* fils. Ceci veut donc dire que les nœuds internes auront tous deux fils. Dans ce type d'arbre, on peut établir une relation entre la taille de l'arbre et le nombre de feuille. En effet, un arbre binaire localement complet de taille n aura *n+1* nœuds. L'arbre suivant est localement complet :

Exemple

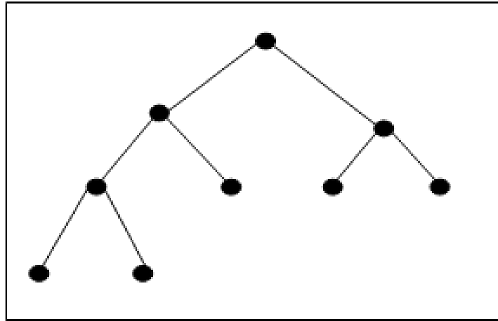


Figure 4.4 – Exemple d'un arbre binaire localement complet.

4.4.3.3 Arbre binaire quasi-complet ou parfait

Un arbre binaire est parfait si tous ses niveaux sont remplis sauf éventuellement le dernier niveau qui a toutes ses feuilles ramenées complètement à gauche.

Exemple

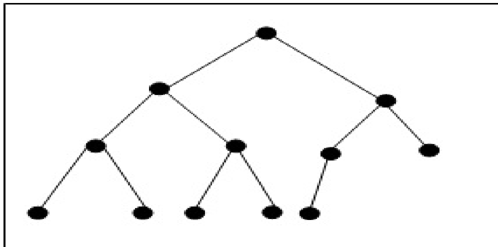


Figure 4.5 – Exemple d'un arbre binaire parfait.

4.4.3.4 Arbre binaire dégénéré ou filiforme

Arbre binaire dégénéré ou filiforme est un arbre formé uniquement des points simples.

Exemple

4.4.3.5 Arbre binaire peigne

Un arbre binaire peigne gauche (resp. droit) est un arbre binaire localement complet dans lequel tous les fils droits (resp. gauche) sont des feuilles.

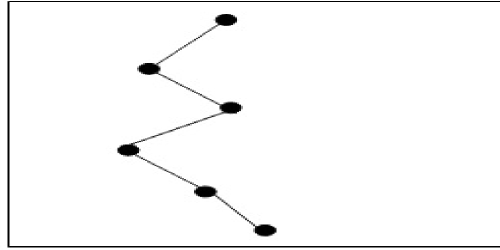


FIGURE 4.6 – Exemple d'un arbre filiforme.

Exemple

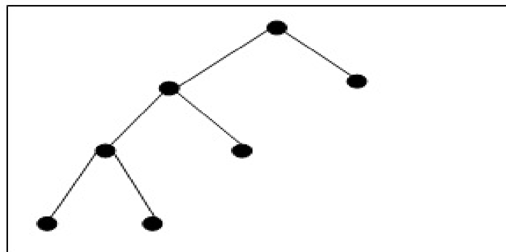


FIGURE 4.7 – Exemple d'un arbre binaire peigne gauche.

4.4.4 Transformation d'un arbre n-aire en un arbre binaire

Un arbre binaire peut toujours être formé à partir d'un arbre n-aire en utilisant les liens premiers fils, frère immédiat. Pour chaque nœud de l'arbre n-aire, le fils gauche de l'arbre binaire est donné par le premier fils dans l'arbre n-aire et le fils droit par le premier frère de l'arbre n-aire.
Exemple

4.4.5 Occurrence et numérotation hiérarchique

Pour désigner un nœud d'un arbre, on lui associe un mot formé de symboles "0" et "1", qui décrit le chemin de la racine de l'arbre à ce nœud, ce mot est appelé occurrence du nœud dans l'arbre. Par définition, l'occurrence de la racine d'un arbre est le mot vide ξ. Si un nœud a comme occurrence le mot μ, son fils gauche a pour occurrence μ_0 et son fils droit a pour

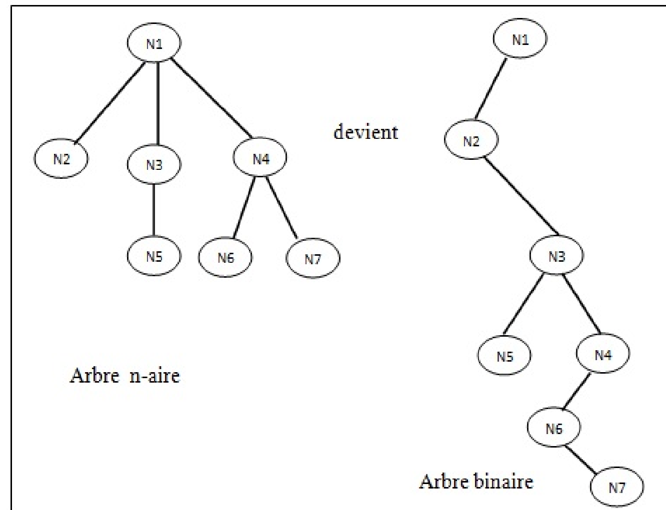


FIGURE 4.8 – Transformation d'un arbre n-aire en un arbre binaire.

occurrence μ_1.
L'intérêt de cette notation est de permettre le codage d'un arbre par un ensemble de mot.
Exemple
Si on a l'ensemble des mots suivant : $\{\ \xi$, 0, 1, 00, 10, 11, 001, 110, 0010, 0011$\}$.
L'arbre équivalent est le suivant :

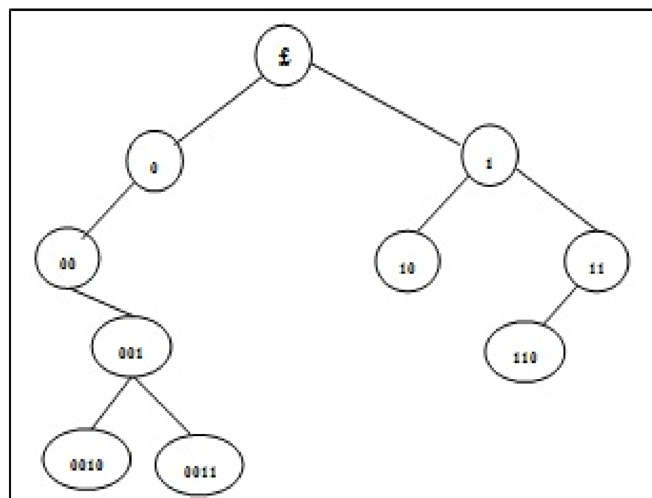


FIGURE 4.9 – Exemple montrant le codage d'un arbre par un ensemble de mot.

Dans les arbres binaires complets, on peut travailler avec la numérotation hiérarchique des nœuds. Cette numérotation est assurée en associant (en ordre croissant et à partir de 1) un numéro à chaque nœud, à partir de la racine : niveau par niveau et de gauche à droite pour chaque niveau.

Exemple

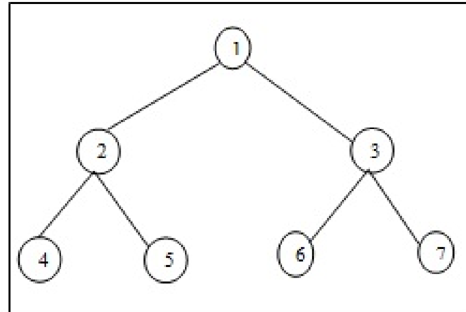


FIGURE 4.10 – Exemple montrant la numérotation hiérarchique d'un arbre complet.

4.4.6 Implémentation d'un arbre binaire

Comme toutes les SD déjà vues, on peut faire l'implémentation d'un arbre, soit en statique, soit en dynamique.

1. **En statique**, par l'utilisation d'un tableau indexé où chaque élément du tableau possède trois champs :
   - Un champ pour l'information effective du nœud,
   - Un champ contenant l'indexe où se trouve le fils gauche,
   - Un champ contenant l'indexe où se trouve le fils droit.
2. **En dynamique** ou en représentation chainée d'où une zone mémoire est allouée pour chaque nœud de l'arbre, cette zone contenant la valeur associée au nœud et deux pointeurs (adresses) vers les fils gauche et droit.

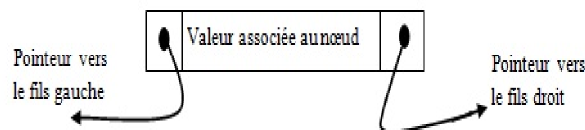


FIGURE 4.11 – Représentation chaînée d'un nœud dans un arbre.

La déclaration d'un arbre implémenté en dynamique se fait de la manière suivant (par exemple, pour un arbre des entiers) :

type Arbre = ^ nœud
    nœud = enregistement
        info : entier;

Fgauche : Arbre ;
Fdroit : Arbre ;
fin ;

Une fois le type Arbre est défini, on peut déclarer une variable de ce type de la manière suivante :
var A : Arbre ;

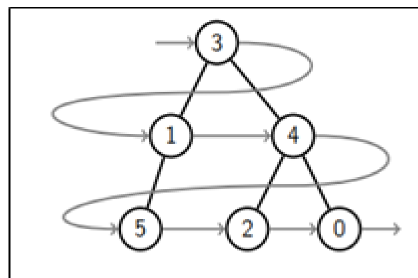
4.4.7 Opérations de base sur les arbres binaires

La structure d'arbre n'est pas liée à des primitives de manipulations caractéristiques, comme **Empiler** et **Dépiler** pour les **piles** ou **Enfiler** et **Défiler** pour les **files**. On distingue, généralement, quelques opérations usuelles de manipulation des arbre comme : le **parcours** d'un arbre, calculer le **degré** d'un arbre, vérifier si un nœud donné est une **feuille** ou non, etc. Il est évidemment possible de faire des opérations de mise à jour comme la suppression ou l'ajout d'un nœud dans un arbre.

4.4.8 Parcourir un arbre

Parcourir un arbre consiste à visiter dans un certain ordre tous ses nœuds, c'est-à-dire énumérer les nœuds de l'arbre. On distingue :

- **Parcours en largeur** : dans un parcours en largeur, on énumère les nœuds par ordre croissant de profondeur des nœuds. Autrement dit, de haut en bas, niveau par niveau (et de gauche à droite).
  Exemple L'ordre de visite des nœuds pour ce parcours est le suisvant : 3, 1, 4, 5, 2, 0.



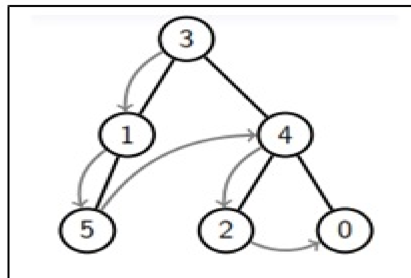
- **Parcours en profondeur à main gauche** : consiste à tourner autour de l'arbre en suivant le chemin qui part à gauche de la racine, et va toujours le plus à gauche possible, en suivant l'arbre et en faisant figurer les sous-arbres vides. Dans le parcours en

profondeur à main gauche, on distingue comme cas particuliers trois ordres classiques :

- **Ordre préfixe, préordre (NGD)** : tout nœud est suivi des nœuds de son sous-arbre Gauche, puis des nœuds de son sous-arbre Droit.

  Exemple

  L'ordre de visite des nœuds pour ce parcours est le suivant : 3, 1, 5, 4, 2, 0.



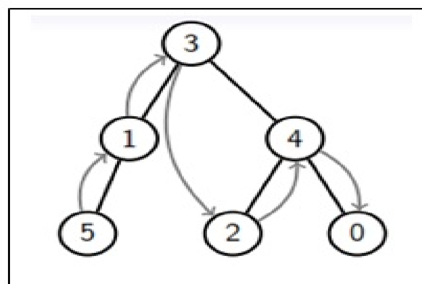
La procédure qui correspond à cet ordre est la suivante :

procedure ordre_ prefixe (A : Arbre) ;
debut
| **si** (A < > Nil) **alors**
| | Ecrire (A^.info) ;
| | ordre_ prefixe (A^.Fgauche) ;
| | ordre_ prefixe (A^.Fdroit) ;
| | **finsi ;**
fin ;

- **Ordre infixe, symétrique (GND)** : tout nœud est précédé des nœuds de son sous-arbre Gauche, puis suivi des nœuds de son sous-arbre Droit.

  Exemple

  L'ordre de visite des nœuds pour ce parcours est le suivant : 5, 1, 3, 2, 4, 0.

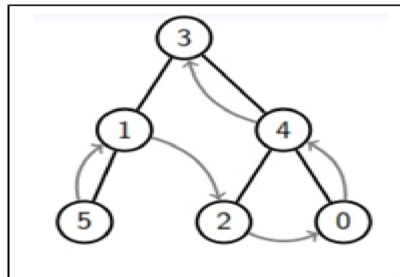


La procédure qui correspond à cet ordre est la suivante :

procedure ordre_ infixe (A : Arbre) ;
debut
| **si** (A < > Nil) **alors**
| | ordre_ infixe(A^.Fgauche) ;
| | Ecrire (A^.info) ;

| | ordre_ infixe (A^.Fdroit) ;
| | **finsi ;**
fin ;

- **Ordre suffixe, postfixe, postordre (GDN)** : tout nœud est précédé des nœuds de son sous-arbre Gauche, et des nœuds de son sous-arbre Droit.
  Exemple
  L'ordre de visite des nœuds pour ce parcours est le suivant : 5, 1, 2, 0, 4, 3.



La procédure qui correspond à cet ordre est la suivante :
procedure ordre_ suffixe (A : Arbre) ;
debut
| **si** (A < > Nil) **alors**
| | ordre_ suffixe(A^.Fgauche) ;
| | ordre_ suffixe (A^.Fdroit) ;
| | Ecrire (A^.info) ;
| | **finsi ;**
fin ;

4.4.8.1 Calculer le degré d'un arbre

Le but est de calculer le degré d'un arbre A. Par définition, le degré d'un arbre représente le nombre des nœuds qui le composent.
Les instructions réalisant cette tâche sont regroupées dans la fonction suivante :
fonction degre_arbre (A :Arbre) : entier ;
debut
| **si** (A = Nil) **alors**
| | degre_arbre ← 0 ;
| | **sinon**
| | degre_arbre ← 1+ degre_arbre (A^.Fgauche) + degre_arbre (A^.Fdroit) ;
| | **finsi ;**
fin ;

4.4.8.2 Savoir si un nœud est une feuille

Le but est de savoir si un nœud donné représente une feuille ou non. Le résultat retourné, donc, est un booléen. Son initialisation se fait au niveau du programme principal à faux.

```
procedure est_feuille (A : Arbre, var verif : booleen) ;
debut
|    si ((A^.Fgauche = Nil) et (A^.Fdroit = Nil)) alors
|      | verif ← vrai ;
|      | finsi ;
fin ;
```

4.5 Applications importantes des arbres

L'arbre est une structure de données fondamentale en informatique, il est très utilisé dans tous les domaines, parce que bien il est adapté à la représentation naturelle des informations homogènes organisées, et il est d'une grande commodité et rapidité de manipulation. L'usage des arbres est multiple car il capte l'idée de hiérarchie. À titre d'exemples, nous pouvons citer :

4.5.1 Le calcul arithmétique

Un arbre de calcul arithmétique est un arbre dont tous les nœuds non feuilles ont exactement deux fils dont les données sur les feuilles sont des nombres et les données sur les nœuds non feuilles sont des signes d'opération.
Considérer les exemples ci-dessous d'évaluations des expressions arithmétiques, pour voir comment sont utilisés les arbres binaires pour représenter les données dans une forme hiérarchique.

4.5.2 Le codage de Huffman

En général, dans un programme, on représente un ensemble d'éléments en affectant un code unique à chaque élément. Par exemple, dans le code standard *ASCII* (American Standard Code for Information Interchange), chaque caractère est représenté par 8 bits. Il faut un nombre minimum de bits pour représenter de façon unique chaque caractère. De façon générale, Il faut $log(n)$ bits pour représenter n valeurs différentes. On parle dans ce cas de codification

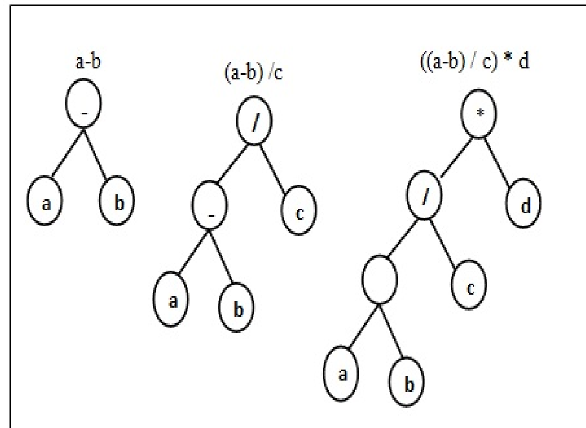


FIGURE 4.12 – Trois exemples montrant l'évaluation des expressions arithmétiques via des arbres.

de longueur fixe. Cette méthode serait la plus efficace si tous les caractères étaient utilisés de façon équitable. Une autre approche consiste à trouver des codes de longueur variable. Cette codification tient compte de la fréquence d'apparition d'un caractère. Il est évident que les caractères les plus fréquemment utilisés devraient avoir des codes plus petits. Cette approche est utilisée par Huffman pour la compression de fichiers. Pour coder ses caractères Huffman commence par construire un arbre (appelé l'arbre de Huffman). Le code de chaque caractère est alors déduit de cet arbre.

4.6 Exercice

Soit l'arbre binaire B représenté par l'ensemble suivant des mots :
B = { ξ, 0 , 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101 }.

1. Donnez la représentation graphique de B en étiquetant les nœuds en ordre hiérarchique de X_1 à X_n (n étant le nombre total de nœuds de B).
2. Calculez la taille, la profondeur, la longueur de cheminement de B.
3. Comment appelle t'on B ? Pourquoi ?
4. Donnez la liste des nœuds en parcourant B en profondeur à main gauche, en ordre préfixe puis suffixe.
5. Donnez la représentation chaînée de B, puis faites sa déclaration.
6. En utilisant la représentation faites en cinquième question, écrivez trois sous-programmes permettant, à partir d'un nœud donné, d'obtenir son père, son fils gauche et son fils droit.

Chapitre 5

Etude de quelques méthodes de tri et de recherche

5.1 Introduction

La majorité des applications informatiques ont besoin de stocker des données en grande quantité. Avant de traiter une donnée, il faut savoir la rechercher pour la retrouver rapidement. Le tri des données est une solution fondamentale pour accélérer la recherche d'une donnée particulière dans un domaine vaste. Dans ce chapitre, nous allons discuter quelques méthodes de tri et de recherche les plus répandues dans la littérature. Pour chaque méthode, nous allons présenter son principe ainsi que son algorithme correspondant.

5.2 Méthodes de tri

Le but d'un tri est d'ordonner les éléments d'un ensemble E en ordre croissant (ou décroissant).
Dans ce qui suit, nous allons procéder de la manière suivante : on commence par une famille E $= < e_1, e_2, e_3, .., e_n >$ stockée sous forme d'un tableau TE (Tableau en Entrée), et on souhaite le permuter de sorte que le tableau TS (Tableau en Sortie) qui en résulte soit ordonné. Afin d'économiser la mémoire et le temps de copie, c'est un seul tableau qui est utilisé, qui au début représente TE et qui finit par représenter TS.
Dans les sous-sections ci-dessous, nous allons présenter les méthodes de tri les plus connues dans la littérature :

5.2.1 Tri par sélection

Le principe de cette méthode est le suivant :

- **Principe**
  C'est sans doute la méthode de tri la plus élémentaire. On cherche le plus petit élément e_m parmi $< e_1, e_2, e_3, ..e_n >$, puis on le place au début du tableau. Cette opération est répétée pour le tableau restant jusqu'à ce que tous les éléments soient dans l'ordre.
- **Algorithme**
  L'algorithme Tri_Selection résume le principe de cette méthode. Nous allons travailler sur un tableau comportant cent éléments entiers.

```
algorithme Tri_Selection ;
const TD = 100 ;
type tab = tableau [1..TD] de entier ;
var T : tab ; TR : entier ;
procedure Saisie (var A : tab ; n : entier) ;
var i : entier ;
debut
|    pour ( i allant de 1 à n) faire
|     | Lire (A[i]) ;
|     | finpour ;
fin ;
procedure Affichage ( A : tab ; n : entier) ;
var i : entier ;
debut
|    pour ( i allant de 1 à n) faire
|     | Ecrire (A[i]) ;
|     | finpour ;
fin ;
procedure Pro_tri_Selection (var A : tab ; n : entier) ;
var X, i, j : entier ;
debut
|    pour ( i allant de 1 à n -1) faire
|     | pour(j allant de i+1 à n ) faire
|     |     | si (A [j] < A[i] ) alors
|     |     |     | X ← A[j] ;
|     |     |     | A [j] ← A[i] ;
|     |     |     | A [i] ← X ;
|     |     | finsi ;
|     | finpour ;
|     |  finpour ;
```

```
fin ;
debut (PP)
|    repeter
|     | Lire (TR) ;
|    Jusqu à ((1 =< TR ) et (TR =< TD)) ;
|     Saisie (T, TR) ;
|     Pro_Tri_Selection (T, TR) ;
|     Affichage (T, TR) ;
fin.
```

5.2.2 Tri par transposition "bulle"

Le principe de cette méthode est le suivant :

- **Principe**
  Une variante du tri par sélection est le tri par transposition, aussi appelé le tri à bulle. On regarde si deux éléments voisins sont dans le mauvais ordre, si oui on les échange. Cette opération est itérée jusqu'à ce que tous les éléments soient dans l'ordre.
- **Algorithme**
  L'algorithme suivant Tri_Bulle réalise le tri par transposition.

```
algorithme Tri_Bulle ;
const TD = 100 ;
type tab = tableau [ 1..TD] de entier ;
var T : tab ; TR : entier ;
procedure Pro_tri_Bulle (var A : tab ; n : entier) ;
var X, i, j : entier ;
debut
|    pour (i allant de 1 à n -1) faire
|     | pour(j allant de 1 à n-i ) faire
|     |     | si (A [j] > A [j+1] ) alors
|     |     |     | X ← A[j] ;
|     |     |     | A [j] ← A[j+1] ;
|     |     |     | A [j+1] ← X ;
|     |     | finsi ;
|     | finpour ;
|     |  finpour ;
fin ;
debut (PP)
|    repeter
|     | Lire (TR) ;
```

```
|    Jusqu à ((1 = < TR ) et (TR = < TD)) ;
|     Saisie (T, TR) ;
|     Pro_Tri_Bulle (T, TR) ;
|     Affichage (T, TR) ;
fin.
```

5.2.3 Tri par insertion

Son principe est le suivant.

- **Principe**

  Cette méthode consiste à piocher une à une les valeurs du tableau et à les insérer, au bon endroit, dans le tableau trié constitué des valeurs précédemment piochées et triées. Les valeurs sont piochées dans l'ordre où elles apparaissent dans le tableau.

  Soit k l'indice de la valeur piochée, les (*k-1*) premières valeurs du tableau constituent le tableau trié dans lequel va être inséré la k^{ime} valeur. Au début de l'algorithme, il faut considérer que le tableau constitué du seul premier élément est trié : c'est vrai puisque ce tableau ne comporte qu'un seul élément. Ensuite, on insère le second élément (k=*2*), puis le troisième (k=*3*), etc. Ainsi, k varie de *2* à n, où n est le nombre total d'éléments du tableau.

- **Algorithme**

  L'algorithme Tri_Insertion regroupe l'ensembles des instructions réalisant cette méthode.

```
algorithme Tri_Insertion ;
const TD = 100 ;
type tab = tableau [ 1..TD] de entier ;
var T : tab ; TR : entier ;
procedure Pro_tri_Insertion (var A : tab ; n : entier) ;
var X, i, j : entier ;
debut
|    pour ( i allant de 2 à n) faire
|     | X ← A[i] ;
|     | j ← i-1 ;
|     | TQ( ( j > 0 )et ( A[j] > X ) ) faire
|     |      | A [j+1] ← A[j] ;
|     |      | j ← j-1 ;
|     |      | A [j+1] ← X ;
|     | finTQ ;
|     |  finpour ;
fin ;
```

```
debut (PP)
|    repeter
|     | Lire (TR) ;
|    Jusqu à ((1 = < TR ) et (TR = < TD)) ;
|     Saisie (T, TR) ;
|     Pro_Tri_Insertion (T, TR) ;
|     Affichage (T, TR) ;
fin.
```

5.3 Méthodes de recherche

Les algorithmes de recherche peuvent varier selon plusieurs critères :

- S'agit-il d'une recherche *interne* (dans la mémoire centrale RAM) ou d'une recherche *externe* (sur le disque dur de l'ordinateur).
- La clé de la recherche est-elle *simple* (un numéro) ou *composée* (un nom + un prénom + une date de naissance) ?
- La structure de données utilisée : tableau, liste linéaire, liste doublement chaînée, etc.
- La nature de la recherche : est-ce qu'on recherche *une occurrence*, la *première occurrence*, *toutes les occurrences* des éléments ayant la même clé,etc.

Dans notre cours, on s'intéresse à une recherche interne sur la première occurrence d'une clé simple dans un tableau à une seule dimension. Etant donné un tableau T de TR éléments (des entiers, par exemple) et un élément *valeur* (à rechercher). Trouver la position (ou l'indice) d'un élément quelconque du tableau *T[1..TR]* ayant la même valeur que l'élément valeur (c'est-à-dire, T[position] = valeur).

Entrée : un entier TR, un tableau T[1..TR] de TR éléments entiers et un élément valeur (un entier).

Sortie : une position (un entier) d'un élément dans le tableau T[1..TR] tel que T [position] = valeur, ou bien -1 si l'élément valeur ne figure pas dans le tableau T[1..TR].

Dans ce qui suit, on présentera les deux algorithmes de recherche les plus connus dans la littérature :

- Recherche séquentielle.
- Recherche dichotomique.

5.3.1 Recherche séquentielle

Cette méthode est très simple, on parcourt le tableau du début à la fin en comparant chaque élément rencontré à la clé recherchée. On peut parcourir tout le tableau ou stopper la recherche dès que la clé soit trouvée.
Cette méthode permet d'effectuer une recherche dans un tableau **ordonné ou non** (relation d'ordre sur les entiers, ordre alphabétique sur les chaînes de caractères, etc). Si le tableau est ordonné, on peut stopper la recherche si la clé n'a pas été trouvée et que l'élément courant du tableau est supérieur (pour l'ordre croissant) à la clé selon la relation d'ordre considérée.
Dans la fonction **Fct_ Rech_ Sequentiel**, lorsqu'on visite un élément A[i] (A représente l'alias de T dans la fonction) d'indice i, on le compare avec val (val représente l'alias de valeur dans la fonction). Si A[i] $= val$, on s'arrête et on retourne i, sinon on continue le parcours du tableau (on incrémente la valeur de i).

```
algorithme Rech_Sequentiel ;
const TD = 100 ;
type tab = tableau [ 1..TD] de entier ;
var T : tab ; TR, Valeur, Position : entier ;
fonction Fct_ Rech_ Sequentiel (A : tab ; n, val : entier) : entier ;
var Pos, i : entier ;
debut
|     Pos ← -1 ;
|     i ← 1 ;
|    TQ (( i < = n) et (Pos = -1 )) faire
|     |     si(A[i] = val) alors
|     |      | Pos ← i ;
|     |      sinon
|     |      | i ← i + 1 ;
|     |      finsi ;
|     |  finTQ ;
|     Retourner(Pos) ;
fin ;
debut (PP)
|    repeter
|     | Lire (TR) ;
|    Jusqu à ((1 = < TR ) et (TR = < TD)) ;
|     Saisie (T, TR) ;
|     Lire (Valeur) ;
|     Position ← Fct_ Rech_ Sequentiel (T, TR, Valeur) ;
|    si ( Position < > -1 )  alors
|     | Ecrire ("la valeur recherchée existe dans la position ", Position) ;
```

```
|    sinon
|      | Ecrire ("la valeur recherchée n"existe pas") ;
|    finsi ;
fin.
```

5.3.2 Recherche dichotomique

La recherche séquentielle se révèle assez inefficace si le nombre d'éléments dans le tableau est élevé. La recherche peut être considérablement accélérée si le tableau est ordonné dans le sens que T[1] $<$ T[2] $<$...$<$ T[TR]. Cette méthode est un exemple classique du paradigme diviser pour régner : on compare d'abord l'élément cherché *valeur* avec T[m] au milieu du tableau, puis on continue la recherche sur la moitié adéquate du tableau.
Dans la fonction **Pro_ Rech_ Dicho**, on compare l'élément recherché **val** avec l'élément A[m] qui se trouve au milieu du tableau initial A. Si A[m] = val, on retourne l'indice du milieu du tableau (c-à-d, m). Par contre, si A[m] $<$ val, on refait le même processus dans le tableau A[...m - 1]. Dans le cas A[m] $>$ val, on refait le même processus dans le tableau A[m + 1...n].

```
algorithme Rech_Dicho ;
const TD = 100 ;
type tab = tableau [ 1..TD] de entier ;
var T : tab ; TR, Valeur, Debu, Position : entier ;
fonction Fct_ Rech_ Dicho (A : tab ; d, f, val : entier) : entier ;
var m : entier ;
debut
|    si (d > f ) alors
|      | Retourner (-1) ;
|     sinon
|      | m ← (d + f) div 2 ;
|      | si(A[m] = val ) alors
|      |      | Retourner (m) ;
|      | sinon
|      |      | si (A[m] > val) alors
|      |      |      | Retourner (Pro_ Rech_ Dicho (A, d, m-1, val)) ;
|      |      | sinon
|      |      |      | Retourner (Pro_ Rech_ Dicho (A, m+1, f, val)) ;
|      |      | finsi ;
|      | finsi ;
|     finsi ;
fin ;
```

```
    debut (PP)
|   repeter
|     | Lire (TR) ;
|   Jusqu à ((1 = < TR ) et (TR = < TD)) ;
|     Saisie (T, TR) ;
|     Lire (Valeur) ;
|     Debu ← 1 ;
|     Position ← Fct_ Rech_ Dicho (T, Debu, TR, Valeur) ;
|   si ( Position < > -1 )  alors
|     | Ecrire ("la valeur recherchée existe dans la position ", Position) ;
|   sinon
|     | Ecrire ("la valeur recherchée n"existe pas") ;
|   finsi ;
fin.
```

5.4 Exercice

Donner la preuve de chaque algorithme présenté dans ce chapitre.

Conclusion générale

DANS ce support de cours, nous avons présenté les structures de données de base en informatique (linéaire et hiérarchique).
Nous avons commencé par les structures de données linéaires d'où nous avons introduit les structures de données listes en général et ses deux formes restreintes : les piles et les files. Ainsi, nous avons exhibé l'importance de ces structures à travers quelques exemples d'applications. Par la suite, nous avons décrit la structure arbre ainsi que quelques opérations classiques que l'on peut effectuer sur un arbre binaire. On a choisi d'écrire ces opérations de manière récursive, car c'est la manière la plus simple d'aborder une telle structure de données.

Le tri des données est une solution fondamentale pour accélérer la recherche d'une donnée particulière dans une structures. A la fin de ce support de cours, nous avons discuté quelques méthodes de tri et de recherche les plus répandues dans la littérature.

Bibliographie

1. T-H. Cormen, C-E. Leiserson, R- L. Rivest " Introduction to Algorithms ", The MIT Press and McGraw-Hill Book Company, 1989.
2. C. Froidevaux, M-C. Gaudel, M. Soria "Types de données et algorithmiques", McGraw-Hill Book Company, 1990.
3. H. Cherroun "Polycopie de cours d'Algorithmes et Type de Données Abstraits", Université de Laghouat, 2010.
4. P. Tellier "Algorithmique et structures de donnée", Département d'Informatique de l'Université Louis Pasteur, 2006.
5. A. Djeffal "Cours d'Algorithmique et structures de données 1", Université Mohamed Khider-Biskra, 2013.
6. "algorithmique, Chapitre 10 : listes chainées", DVD-MIAGE de Nantes, France (http ://miage.univ-nantes.fr/miage/DVD-MIAGEv2/Algo.html), dernière consultation 10/08/2015.
7. "algorithmique, Chapitre 11 : piles et files", DVD-MIAGE de Nantes, France (http ://miage.univ-nantes.fr/miage/DVD-MIAGEv2/Algo.html), dernière consultation 10/08/2015.
8. "Chapitre 6 : Structures de données listes et algorithmes" (http ://www.google.dz/url ?sa=t&rct=j&q=&esrc=s&source=web&cd=3&cad=rja&uact=8&ved=0CCwQFjACahUKEwiCxbHtm7PHAhX IbhQKHdpRAA8&url=http%3A%2F%2Ffdrouillon.free.fr%2Falldocs%2F_LivreC_CPP%2FChap6_%2520Strct%2520donnees%2520 listes %2520%26%2520algo.pdf &ei=iHPTVYLhOcjdUdqjgXg&usg=AFQjCNG8hDhJGLV9MQhOGlvLpt1zM3LpIQ&sig2=LF1B6YVr7lIA7L9XGK2YhQ), dernière consultation 18/08/2015.
9. "Cours 4 et 5 : Les arbres", IUT De Villetaneuse, Département informatique, Université Paris 13, France, 2005.
10. M. Bouzenada "Cours : Type Abstrait des Données (TAD) / Algorithmique", deuxième année ingénieur en informatique, Université de Jijel, 2000-2001.
11. O. Bournez "Cours 3 : Arbres_ Parcours / Algorithmique", LIX Ecole Polytechnique, 2011.
12. A.Dargham "Chapitre 5 : Algorithmes de Recherche et de Tri", 3 Année GI, ENSA, Oujda, 2009-2010.